# From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration

Vander Alves*
Informatics Center, UFPE
vander@acm.org

Alberto Costa Neto*
Informatics Center, UFPE
acn@cin.ufpe.br

Sérgio Soares*
Computing Systems Department, UPE
sergio@dsc.upe.br

Gustavo Santos
Meantime Mobile Creations
gustavo.santos@cesar.org.br

Fernando Calheiros
Meantime Mobile Creations
fernando.calheiros@cesar.org.br

Vilmar Nepomuceno
Meantime Mobile Creations
vsn@cesar.org.br

Davi Pires
Meantime Mobile Creations
davi.pires@cesar.org.br

Jorge Leal
Meantime Mobile Creations
jorge.leal@cesar.org.br

Paulo Borba*
Informatics Center, UFPE
phmb@cin.ufpe.br

## ABSTRACT

Apart from adoption strategies, an existing Software Product Line (SPL) implemented using some variability mechanisms can be migrated to use another variability mechanism. In this paper, we present some migration strategies from one SPL implemented with conditional compilation to one using Aspect-Oriented Programming (AOP). The strategies present a variability pattern handled by the first mechanism and shows how it can be translated into a pattern using AOP constructs. We also show and discuss that some variability patterns cannot be migrated into AOP. The discussion centers around a commercial SPL in the mobile games domain.

## 1. INTRODUCTION

Adoption strategies for Software Product Lines (SPL) frequently involve bootstrapping existing products into a SPL (*extractive approach*) and extending an existing SPL to encompass another product (*reactive approach*), or their combination [8, 4]. The *proactive approach*, in which SPL design and implementation is accomplished for all products in the foreseeable horizon, may be less frequent in practice than the former approaches due to its incurred high upfront investment and risks. Extractive and reactive approaches can be enacted by the application of *program refactorings*.

Apart from adoption strategies, there may be a case when there is an existing SPL already implemented using some variability mechanisms and we would like to implement it using another variability mechanism. We refer to the process of accomplishing this as *migration strategy*, and reasons for accomplishing it include moving to a mechanism that better supports understandability, traceability, and further evolution of the SPL in the reactive scenario.

In this paper, we present some migration strategies from one SPL implemented with conditional compilation to one using Aspect-Oriented Programming (AOP) [7]. The strategies present a variability pattern handled by the first mechanism and shows how it can be translated into a pattern using AOP constructs. We also show and discuss that some variability patterns cannot be migrated into AOP. The discussion centers around a commercial SPL in the mobile games domain.

Section 2 presents the case study used throughout the rest of the paper, motivating the need for migration strategies. Next, Section 3 presents migration strategies. Section 4 then addresses some mappings not possible in this migration strategy. Related work is considered in Section 5, and concluding remarks offered in Section 6.

## 2. THE CASE STUDY

The goal of the case study was to define and evaluate migration strategies for a mobile game SPL. The SPL considered was *Ronaldinho Total Control*[1], where player controls a soccer player in order to get the timing to keep the ball bouncing, make sequences of perfect hits to get bonuses, and get items to make his task easier and achieve the highest score. The game is running in a number of different devices. Devices differ in issues such as memory, screen sizes, additional keys, processing power, which ultimately constrains the features available in each of them. Thus, there are a number of versions of the game running in each device, where each version has slightly different features. The number of instances of this SPL is 16. Figure 1 illustrates the game screen of the game running in two of these different devices.

In Figure 1, the screen on the right is from a power end device, whereas the screen on the left is from a resource-constrained device. Apart from screen dimensions, we can also notice the existence of a bird and a cloud on the right. These are actually scenery objects that move in the background, called *croma* feature. This feature is optional in the SPL and is not present in the device on the left, since due to its constraint on bytecode size.

In terms of implementation, the variabilities involved have different granularity: some relate to the existence or not of particular proprietary drawing API, whereas others happen

*Software Productivity Group http://www.cin.ufpe.br/spg.

[1]Access provided by our industrial partner

**Figure 1: Game screen of the game in different devices.**

within classes involving addition/removal of fields and code blocks inside methods.

The original variability mechanism of the SPL is conditional compilation, which is still largely used in the industry, especially in the mobile games domain. Nevertheless, this mechanism is not an appropriate substitute for proper programming language support. Also, such mechanism has poor legibility and leads to lower maintainability.

The variability of this domain shows considerable tangling and crosscutting. Therefore, it is worth investigating the usefulness of AOP mechanism in handling them. In this context, we propose a number of migration strategies discussed in the next section.

## 3. MIGRATION STRATEGIES

In this section, we present some migration strategies. They were adopted in our case study to migrate from a mobile game SPL implemented using conditional compilation to AspectJ constructs. Some kinds of variations could not be solved using the current version of AspectJ (1.5) and are presented later in Section 4. Each strategy is first described in the context of a concrete example; then it is generalized in a template form, so that it can support automation, which is reagarded as future work.

### 3.1 Super Class Variation

In the case study, there were variations in the super class of some classes. These variations occur, for example, when defining a `Canvas` class that is used to draw shapes and images on the screen. For Nokia devices, it is required that these classes extend the Nokia API class `com.nokia.mid.ui.FullCanvas` instead of MIDP [12] class `javax.microedition.lcdui.Canvas`. If the device supports MIDP 2.0 or is a Siemens mobile device, `Canvas` super classes are also different, called respectively `javax.microedition2.lcdui.game.GameCanvas` and `com.siemens.mp.color_game.GameCanvas`. As a consequence, dealing with this variation requires changing an `import` declaration and the corresponding super class name in the `extends` clause. Using conditional compilation tags, it is possible to define a different import and extends declaration for each of those variations. The following piece of code shows how this variability mechanism is employed to address such variations for configurations corresponding to Nokia and MIDP devices.

```
//#ifdef nokia_device
//#  import com.nokia.mid.ui.FullCanvas;
//#else
//#  import javax.microedition.lcdui.Canvas;
//#endif
...
//#ifdef nokia_device
//#  public class MainCanvas extends FullCanvas {
//#else
//#  public class MainCanvas extends Canvas {
//#endif
...
```

Using AspectJ, such variability can be addressed by declaring an aspect for each possible super class alternative, corresponding to a different configuration. A `declare parents` clause with the required class name is defined in the aspect. Additionally, the corresponding import declaration is transferred to the aspect. The piece of code below shows the result of applying this strategy to the example above.

```
//core
public class MainCanvas {...}

//Nokia configuration
import com.nokia.mid.ui.FullCanvas;
public aspect NokiaCanvasAspect {
  declare parents: MainCanvas extends FullCanvas;
  ...
}

//MIDP configuration
import javax.microedition.lcdui.Canvas;
public aspect MIDPCanvasAspect {
   declare parents: MainCanvas extends Canvas;
   ...
}
```

The approach presented above only works because `FullCanvas` is a subclass of `Canvas`, which is a precondition of declare parents. The classes `GameCanvas` (MIDP 2.0 and Siemens) also respect this rule.

This strategy can be generalized by a pair of source and target templates specifying a transformation on code assets of the SPL. The source template is as follows:

```
//#ifdef TAG
//#  ts'
//#else
//#  ts''
//#endif

//#ifdef TAG
//#  public class C extends C' {
//#else
//#  public class C extends C'' {
//#endif
       fs
       ms
     }
```

Where `TAG` is a conditional compilation tag, whose selection in the SPL configuration binds the superclass of `C` to `C'`, including the corresponding import. When not selected in the SPL configuration, the superclass of `C` is bound to `C''`, also including its corresponding import. We denote the set of type declarations by `ts'` and `ts''`. Also, `fs` and `ms`

denote field declarations and method declarations, respectively.

Code assets matching the source template are transformed according to the following target template, where aspect `A` binds the superclass of `C` to `C'`. The import required by `C'` is in `ts'` and is moved aspect `A`.

```
//core
public class C {
  fs
  ms
}

//configuration 1
ts'
public aspect A {
  declare parents: C extends C';
}

//configuration 2
ts''
public aspect B {
  declare parents: C extends C'';
}
```

## 3.2 Interface Implementation Variation

Another kind of variation in hierarchy addressed in the case study was to make a class implement a different interface. It usually happens due to the use of different APIs requiring the implementation of specific interfaces. This variability issue is similar to the one presented in the previous subsection and can be handled similarly in the migration strategy. The main difference is that it uses `declare implements` instead of `declare parents`.

## 3.3 If Condition Variation

A common variation in mobile devices is the number and type of keys in the keypad. Additionally, the values that represent key pressing events differ between mobile devices families. This latter variability is usually implemented through blocks of constant definitions with different values subject to conditional compilation. Other possible implementations include macro and configuration files.

When migrating to AspectJ, it is possible to introduce constants via inter-type declarations with the appropriated values. Additionally, there are variations in `if` conditions responsible for checking whether an specific key has been pressed and launch the code that treats the event. These variations usually required to add more `or-conditions` to treat the additional keys. The following code shows an example of this situation.

```
public class MainCanvas extends Canvas {
  protected void keyPressed(int keyCode) {...
    if (keyCode == LEFT_SOFT_KEY
//#ifdef device_keys_motorola
//#     ||  keyCode == -softKey
//#endif
    ) {
      // handle key event
    }
    ...
  }
}
```

The previous example shows that an additional `or-condition` can activate the code inside `if` command for Motorola mobile devices. With conditional compilation, using one or more `ifdef`'s addresses this variability issue.

We defined a migration strategy that involved 1) the extraction of `if-condition` to a new method defined in the class containing the base condition; 2) the use of an around advice in an aspect to enhance the base condition. The result is as follows:

```
public class MainCanvas extends Canvas {
  protected void keyPressed(int keyCode) {...
    if(compareEquals(keyCode, softkey)) {
       // handle key event
    }
  }
  private boolean compareEquals(int keyCode,
                                int softKey) {
    return keyCode == softKey;
  }...
}

//Motorola device configuration
public privileged aspect DeviceKeysMotorola {
  boolean around(int keyCode, int softKey) :
    execution(private boolean MainCanvas.compareEquals(..))
    && args(keyCode, softKey)
  {
    return keyCode == softKey || keyCode == -softKey;
  }
  ...
}
```

The source template of the migration strategy is shown next:

```
ts
public class C {
 fs
 ms
 T m(ps) {
   body
   if (cond
//#ifdef TAG
//#    op cond'
//#endif
    ) {
      body'
    }
    body''
  }
}
```

where `cond` represents the base condition and the variation is an additional expression `op cond'`. The expression `op` represents binary operators and `cond'`, any boolean expression. Also, `body`, `body'`, and `body''` denote blocks of statements in a method. The target template of this strategy is presented next:

```
ts
public class C
 fs
 ms
 T m(ps) {
   body
   if (getCond(ps')) {
     body'
   }
   body''
 }
 boolean getCond(ps') {
   return cond;
 }
}

//SPL configuration handling variability issue
public aspect A {...
  boolean around(ps') :
    execution(boolean C.getCond(..))
    && args(ps')
  {
    return cond || cond';
  }
}
```

It is important to notice that using an `around-advice` allows substituting or complementing the original condition specified in the `if` statement, by executing or not a `proceed` statement.

## 3.4 Feature Dependency

This section presents the strategy employed to migrate a feature depending on others features. In the case study, there is a feature called Arena, that allows posting game results to a public server for ranking purposes. This feature also presents results on the device screen. Since screen size is variable across devices, it would be necessary to develop an Arena feature to each appropriated screen size. Using conditional compilation, this feature implementation is spread in many classes and tangled with other functionalities.

In the following code, if the tag `feature_arena_enabled` is enabled during SPL instantiation, some common constants to paint the scroll bar are defined, but the constants `ARENA_SCROLL_HEIGHT` and `ARENA_SCROLL_POS_Y` have different values depending on the device's screen size.

```
public class MainScreen {
//#if feature_arena_enabled
   /** Constants to paint the scroll bar */
    //#if device_screen_128x128
    //#  public static final int ARENA_SCROLL_HEIGHT = 92;
    //#  public static final int ARENA_SCROLL_POS_Y = 17;
    //#elif device_screen_128x117
    //#  public static final int ARENA_SCROLL_HEIGHT = 81;
    //#  public static final int ARENA_SCROLL_POS_Y = 16;
    //#endif
//#endif
...
}
```

The strategy adopted to implement this feature dependency was to define an aspect called `ArenaAspect` to handle the core of the feature and, for each screen size variation inside Arena, define others aspects, `ArenaScreen128x128` and `ArenaScreen128x117`. Additionally, there is the following constraint on the SPL configuration knowledge: when the optional feature Arena is enabled, one of the aspects `ArenaScreenWxH` is automatically selected depending on the screen size of the device. The piece of code below shows the result of applying this strategy to the class `MainScreen` mentioned previously.

```
public class MainScreen {... }

public aspect ArenaAspect {
   /** Constants to paint the scroll bar */
}

public aspect ArenaScreen128x128 {
  public static final int
      MainScreen.ARENA_SCROLL_HEIGHT = 92;
  public static final int
      MainScreen.ARENA_SCROLL_POS_Y = 17;
}

public aspect ArenaScreen128x117
  public static final int
      MainScreen.ARENA_SCROLL_HEIGHT = 81;
  public static final int
      MainScreen.ARENA_SCROLL_POS_Y = 16;
}
```

The template generalizing this migration strategy is presented next. It is important to notice that `TAG_A` represents an optional feature and tags `TAG_B1` and `TAG_B2` represent features depending on `TAG_A`.

```
public class C {
  fs
  ms
 ...
//#if TAG_A
//#  fs'
//#  ms'
   //#if TAG_B1
   //#  fs''
   //#  ms''
   //#elif TAG_B2
   //#  fs'''
   //#  ms'''
   //#endif
//#endif
}
```

The target template of this strategy is presented next, where `C.fs'`, `C.fs''` and `C.fs'''` are the sets of fields introduced via inter-type declaration into class `C` by the aspects composed with `C`. The same pattern is used for methods, but they are named `C.ms'`, `C.ms''` and `C.ms'''` instead. Aspect `A` is included in the SPL instance iff feature `A` is selected; aspects `AB1` and `AB2` are present in the SPL instance iff their corresponding features are present and feature `A` is also selected.

```
public class C {
  fs
  ms
}
public aspect A {
  C.fs'
  C.ms'
}
public aspect AB1 {
  C.fs''
  C.ms''
}
public aspect AB2 {
  C.fs'''
  C.ms'''
}
```

## 3.5 Discussion

Some of the strategies presented previously could benefit from general OO techniques (e.g. using abstract methods and subclassing, patterns and so forth), but this would imply having a subclass for each possible device, thus leading to complex class hierarchies. Additionally, many more classes would be involved, thus incurring into a penalty in terms of bytecode size, a critical issue in the mobile application domain.

The strategies replace the scattered ifdefs by a number of aspects, which have to be managed. This is addressed by a configuration knowledge, relating device configurations to configurations involving sets of aspects and core classes. The AO advantage lies in the fact that the extracted variability can be used elsewhere without replicating code, whereas the ifdef variability can only be used in that context.

Although some variabilities addressed are very fine-grained, they are crosscutting, because they can be logically grouped together with other fine-grained variability affectting other join points, such that this unit–the aspect–implements a feature. More generally, we could further cluster crosscutting variability so that it can be more broad in a module-classes and aspects–implementing a given feature. This is regarded as future work.

Some strategies not shown involved handling variability in the definition and usage of constants. The usage of constants can certainly benefit from using final static variables, an appraoch we have used; however, variability in the definition constants themselves was addressed by a migration strategy to use inter-type declaration. On the other hand, mode-driven approach is not appropriate in this domain because device constraints such as memory imply constraints on game features, thus preventing the definition of a pure platform independent model.

# 4. OPEN ISSUES

In addition to the migration strategies already presented, there are some variations for which we could not define a migration strategy using AspectJ. In this section, we address those by showing how AspectJ's current implementation does not support them. In some cases, we provide alternative solution using other approaches; in others, we present candidate extensions to the AspectJ language.

## 4.1 Import Variation

In the performed case study, there are variations between device families that use different APIs. These APIs define types with the same name and the same interfaces to facilitate the porting task. However, those types are defined in different name spaces, since each API has its own package name. For instance, the following piece of code depicts an example of such variation. The code originally written with conditional compilation tags imports a `Sprite` type from `javax.microedition.lcdui.game` package or from `com.meantime.j2me.util.game` depending on the MIDP version it uses. The latter is used when generating a release to device families that use MIDP 2.0, and the former otherwise.

```
//#ifdef game_sprite_api_midp2
//#   import javax.microedition.lcdui.game.Sprite;
//#elif
//#   import com.meantime.j2me.util.game.Sprite;
//#endif
...
```

Since the AspectJ language in its current version (1.5) does not handle variability at the import clauses granularity, there is not a solution to migrate this conditional compilation code to AspectJ code. One alternative for such kind of variations would be extending AspectJ with inter-type declarations that insert an import clause in a type. Another possibility would be using a transformation system [3] that uses generative techniques allowing to control such kind of elements in the source code.

This concrete example can be generalized to variations that demand different imports clauses, regardless of the types' name. The form of such problem is presented in the following piece of code.

```
...
//#if TAG_1
//#   import I_1;
//#elif TAG2
//#   import I_2;
...
//#elif TAG_n
      import I_n;
//#endif
...
```

where `TAG_1`, `TAG_2`, and `TAG_n` are conditional compilation tags that define variation code and `I_1`, `I_2`, and `I_n` are the imports expressions.

## 4.2 Superclass Constructor Call

Another example of conditional compilation code that could not be migrated to AspectJ is a call to a superclass constructor. In this example, two variants demands calling the superclass constructor with the parameter `false` if the device uses MIDP 2.0 or if it is a Siemens device; otherwise, no explicit super call is needed, thus implying an implicit to the empty superclass constructor.

```
...
  public MainCanvas() {
//#if device_graphics_canvas_midp2 ||
//#   device_graphics_canvas_siemens
//#      super(false);
//#endif
          ...
  }
...
```

AspectJ does not support such migration since an advice cannot call a constructor using neither `super` nor `this`. In fact, it is possible to write a code that prevents the superclass constructor to execute, but not a code that executes one constructor instead of another.

A possible solution would be extending AspectJ to allow writing an advice that executes first in a constructor call and can call the superclass constructor or another constructor in the same class, or using the transformation system mentioned before to add such constructor call.

This issue can be generalized to any variation that demands a different superclass constructor call:

```
...
  CONSTRUCTOR(PARS) {
//#if TAG
//#   super(ARGS);
//#endif
      ...
  }
```

or a change in the inline calls of class constructors.

```
...
  CONSTRUCTOR(PARS) {
//#if TAG
//#   this(ARGS);
//#endif
      ...
  }
```

where `PARS` is the constructor parameter list, which can be empty, and `ARGS` is the argument list, possible empty, of the class or superclass constructor call.

## 4.3 Adding an else-if Block

Another migration issue occurs when a variation demands the insertion of new `else-if` blocks in a conditional statement. This case is common with feature variations that add new screens to the game. The code that paints the current screen must check the type of the current screen in a long `if-else-if` structure; therefore, new screen type checks are added as `else-if`'s to the end of this structure.

```
    ...
    if (this.screenMode ==
          Resources.MAIN_SCREEN_MODE_SPLASH) {
      //code
    } else if (this.screenMode ==
          Resources.MAIN_SCREEN_MODE_LOGO) {
      //code
    }
//#ifdef feature_arena_enabled
//#  else if (this.screenMode ==
//#        Resources.MAIN_SCREEN_MODE_ARENA_WELCOME) {
//#    //code
//#  } else if (this.screenMode ==
//#        Resources.MAIN_SCREEN_MODE_ARENA_LOGIN) {
//#    //code
//#  }
//#endif
```

There is no construction in AspectJ that deals with conditional statements or any similar that would address this issue. The alternative would be again using the transformation system to generate the code to be added. An AspectJ extension that intercepts conditional statements does not seam very useful, since the conditional statements are not named, which leads to ambiguity when a method has more than one conditional statement.

This issue can be generalized by the following form:

```
    ...
    if(EXP_1) {
      // code
    } else if (EXP_2) {
      // code
    }
    ...
//#ifdef TAG
    else if(EXP_n) {
      // variation
    }
//#endif
```

where `EXP_1`, `EXP_2`, and `EXP_n` are boolean expressions.

# 5. RELATED WORK

We have previously explored SPL adoption strategies at the implementation level [2] and at the feature model level [1]. In this work, instead of adoption strategies, we address migration of variability mechanism in an existing SPL.

Lopez-Herrejon et al [10] have evaluated the use of different variability mechanisms in providing support for modularization of features. Differently, our work focuses on the migration of one technique to another by providing strategies specified by means of templates. Our work could benefit from theirs by considering migration strategies to other target variability mechanisms in cases where AspectJ does not have appropriate constructs.

Another work [6] explores the application of refactoring to SPL Architectures. They present metrics for diagnosing structural problems in a SPL Architecture, and introduce a set of architectural refactorings that can be used to resolve those problems. These metrics could be useful for detecting bad smells and guiding the application of our migration strategies.

Monteiro et al [11], Laddad [9], and Cole et al [5] discuss refactoring from Java to AspectJ programs. Although these works are not directly related to SPLs, we can use several OO to AO refactorings to extract variations of a mobile application in a extractive approach to define a SPL. In the particular example addressed in this paper, we worked on an SPL that was already implemented using conditional compilation and extracted the variations to use aspects, following an approach which was neither extractive, nor proactive nor reactive.

# 6. CONCLUSION AND FUTURE WORK

We have presented migration strategies from one SPL implemented with conditional compilation to one using AOP. The strategies present a variability pattern handled by the first mechanism and shows how it can be translated into a pattern using AOP constructs. We also show and discuss that some variability patterns cannot be migrated into AOP. The discussion centers around a commercial SPL in the mobile games domain.

As future work, we intend to explore other target variability mechanisms and also to enhance AspectJ to overcome the mappings not currently possible in our migration strategy. We also plan provide automation support for the templates and to asses them in different product lines. Finally, we will explore clustering the variability into a broader context.

# 7. ACKNOWLEDGMENTS

We gratefully acknowledge our industrial partner, Meantime Mobile Creations, for granting us access to the game in this case study. This research was partially sponsored by CNPq (grants 481575/2004-9, 141247/2003-7, 552068/2002-0) and MCT/FINEP/CT-INFO (grant 01/2005 0105089400).

# 8. REFERENCES

[1] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *Proceedings of the 5th ACM International Conference on Generative Programming and Component Engineering (GPCE'06)*. ACM Press, Oct 2006. To appear.

[2] Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and evolving mobile games product lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 70–81. Springer-Verlag, Sep 2005.

[3] Fernando Castor, Kellen Oliveira, Adeline Souza, Gustavo Santos, and Paulo Borba. JaTS: A Java transformation system. In *XV Brazilian Symposium on Software Engineering*, pages 374–379, Rio de Janeiro, Brazil, October 2001.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[5] Leonardo Cole and Paulo Borba. Deriving refactorings for aspectj. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development - AOSD'05*. ACM press, March 2005.

[6] Matt Critchlow, Kevin Dodd, Jessica Chou, and André van der Hoek. Refactoring product line architectures. In *IWR: Achievements, Challenges, and Effects*, pages 23–26, 2003.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, 1997.

[8] Charles Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th Workshop on Software Product-Family Engineering*, pages 282–293, 2001.

[9] Ramnivas Laddad. Aspect oriented refactoring series. In *TheServerSide.com*, December 2003.

[10] Roberto E. Lopez-Herrejon, Don S. Batory, and William R. Cook. Evaluating support for features in advanced modularization technologies. In *ECOOP*, pages 169–194, 2005.

[11] M. P. Monteiro and J. M. Fernandes. Object-to-aspect refactorings for feature extraction. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development - AOSD'04*. ACM press, March 2004.

[12] Java Community Process. *Mobile Information Device Profile 2.0*. http://jcp.org/aboutJava/communityprocess/-final/jsr118/index.html, 2004.